**Лекція № 6. Програми лінійної, розгалуженої та циклічної структури. Загальний огляд. Оператори циклу**

**Пригадаємо.**

Виразом називається мовна конструкція, яка складається з лексем, вираз, який використовується для обчислення значення величини може складатися з одного або декількох операндів, поєднаних знаками операцій, припустимих для величин, що використовуються. Послідовність виконання операцій така:

1) операції в круглих дужках;

2) функції;

3) ! (not);

4) \*, /, && (and);

5) +, -, || (or);

6) =, > , <, <=, >=.

Правила використання виразів:

1. Вираз записується в один рядок.
2. У виразах використовуються тільки круглі дужки, причому кіль­кість дужок, що відкриваються, повинна відповідати кількості дужок, що закриваються.
3. Не можна записувати підряд два знаки арифметичних операцій.
4. Операції одного старшинства виконуються послідовно зліва направо.

Виведення значень виразів ми можемо здійснювати за допомогою операторів потокового введення-виведення **cin** та **cout** або форматованого введення-виведення **scanf** та **printf**.

**Областю дії оголошення змінної, тобто ідентифікатора змінної** є частина програми, в якій його можна використовувати для доступу до зв’язаної з ним області пам’яті. Залежно від області дії змінна може бути локальною або глобальною. Локальна змінна визначена всередині блока (блок розташований між фігурними дужками). Область її дії обмежена початком опису змінної та кінцем блока, включаючи усі вкладені блоки. Змінна, визначена поза будь-яким блоком, називається глобальною, і областю її дії вважається файл, у якому вона визначена від початку опису до його кінця.

При написанні програм використовують такі базові алгоритмічні конструкції :

1. Слідування – блоки алгоритму виконуються послідовно.

2. Розгалуження – в залежності від умови виконується одна або інша гілка алгоритму.

3. Цикл – група блоків алгоритму виконуються декілька разів. Обов’язковою вимогою є наявність умови виходу з циклу.

На попередніх заняттях ми розглядали програми лінійної структури, які мали алгоритмічну структуру слідування. В цій лекції ми розглянемо реалізацію циклу.

**Прості інструкції повторення обчислень**

Циклічні, тобто повторювані, обчислення задають за допомогою оператору циклу. Який має форми:

Інструкція циклу з передумовою **while**,

Інструкція циклу з післяумовою **do while**,

Інструкція циклу **for**.

Оператори циклу використовують для здійснення багаторазового повторення деякої послідовності дій. Кожен цикл складається з тіла циклу, тобто операторів, що виконуються декілька разів. Один прохід циклу називається ітерацією.

Інструкція циклу з передумовою **while** виконується, якщо умова перевіряється до початку циклу, і має вигляд:

**while (умова) *інструкція***

Слово **while** є зарезервованим, дужки обов'язкові, **while (*умова*)** – це **заголовок циклу**, а ***інструкція*** – **тіло**.

Інструкція циклу виконується так. Спочатку обчислюється умова в заголовку. Якщо вона істинна, то виконується тіло циклу та знов обчислюється умова. Якщо вона істинна, то все повторюється. Виконання інструкції циклу закінчується, коли обчислено значення умови **false**, тобто хибність. Отже, в останньому циклі тільки обчислюється умова, а тіло не виконується. Якщо при першому обчисленні умова хибна, то тіло циклу не виконується жодного разу. **Ітерація циклу** складається з перевірки умови циклу та виконання після неї тіла циклу. Інструкції циклу з передумовою відповідає блок-схема на рис. 2.

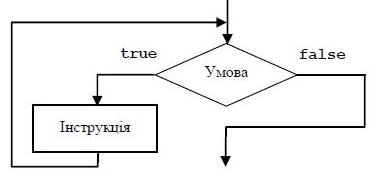


Рис. 2. Блок-схема інструкції циклу з передумовою

Умову в інструкції циклу називають **умовою продовження**, оскільки, якщо вона істинна, то виконання інструкції циклу продовжується. Цикл починається обчисленням умови, тому її ще називають **передумовою**. Інструкції циклу з передумовою застосовують зазвичай тоді, коли кількість повторень циклу наперед невідома.

Приклад: наступна послідовність операторів обчислює суму квадратів перших **n** натуральних чисел:

**int і=0, sum=0; int n=5;**  
**while (і < n) sum += ++і \* і;**

**~~{i=i+1; sum=sum+i\*i;}~~**

**~~sum=sum+(++i)\*i~~**

Інструкція циклу з **післяумовою**, або **do**-інструкція, має загальний вигляд

**do *інструкція* while (умова);**

**Оператор циклу з післяумовою do while** звичайно застосовується у випадках, коли тіло циклу виконується хоча б один раз. Слово **do** (виконувати) є ключовим. Інструкція циклу з післяумовою виконується так: спочатку виконується тіло циклу, потім обчислюється умова. Якщо вона хибна, то цикл завершується, інакше повторюється тіло й знову обчислюється умова. На відміну від інструкції циклу з передумовою, цикл *починається діями в тілі циклу* та закінчується обчисленням умови. Умова перевіряється після виконання тіла циклу, тому її називають **післяумовою**. Тіло циклу, заданого **do**-інструкцією, виконується обов'язково хоча б один раз (на відміну від **while**-інструкції). Інструкцію циклу з післяумовою використовують, коли потрібно спочатку виконати тіло циклу, і лише потім перевіряти умову продовження. Циклу з післяумовою відповідає блок-схема на рис. 3.

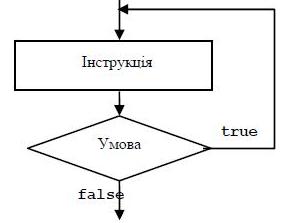


Рис. 3. Блок-схема інструкції циклу з післяумовою

Приклад.Потрібно з клавіатури ввести ціле число від 10 до 99. Якщо користувач набрав число за межами цього діапазону, то слід *повторити спробу*. Отже, спочатку треба вводити число, а потім перевіряти умову того, що число є двозначним.

**do {**

**cout << "Enter one integer in [10,99]>";**

**cin >> k;**

**} while (!(10<=k && k<=99)); // 10<=k && k<=99**

Інструкція циклу **for** або **for**-інструкція, має загальний вигляд

**for (*початкова дія*; *умова*; *перехідна дія*)**

***основна дія***

Слово **for** зарезервоване, дужки та два знаки; усередині дужок є обов'язковими. Початкова дія, умова й перехідна дія є *виразами* (кожен із них може бути порожнім), основна дія – *інструкцією*. Тілом циклу **for** називають його основну дію. Інструкція **for** виконується так само, як і інструкції вигляду

***початкова дія*;**

**while (*умова*)**

**{**

***основна дія*;**

***перехідна дія*;**

**}**

Оператор циклу **for** реалізується таким чином:

* виконується початкова дія - вираз ініціювання (виконання цієї нотації може бути здійснено до оператора **for**);
* обчислюється вираз-умова;
* якщо умовний вираз приймає значення «істина» — виконуються оператори циклу;
* обчислюється вираз ітерації;
* знову перевіряється умова;
* як тільки умова прийме значення **0** («хиба, неправда»), керування передається оператору, що розташований за оператором циклу **for**.

Оскільки в операторі **for** перевірка виразу-умови відбувається перед циклом, то у випадку помилкової умови цикл може жодного разу не виконуватися.

Оператор **for** може використовувати декілька змінних, що керують циклом, а будь-які вирази можуть бути відсутніми, наприклад:

**int n, у;**

**for (int к = 0, n = 20; к <= n; k++, n--)**

**y = k \* n;**

або

**int і=0;**

**for (; і < 4; i++)**

**for (;;) {…};**

Перший фрагмент має два вирази ініціювання і два вирази ітерації. Спочатку відбувається присвоювання значень змінним **k = 0 і** **n = 20**, далі здійснюється порівняння **k <= n** і, якщо ця умова має значення «істина», то буде виконуватися тіло циклу, а потім вираз **k++ і** **n–**, якщо ж умова не виконується, то цикл припиняє свою роботу.

C++ дозволяє поєднати ці дві дії в одному виразі – за допомогою операції послідовного обчислення. Операція зі знаком "**,**" позначає послідовне обчислення виразів, записаних через кому (в прикладі це **к = 0, n = 20;)**. Ця послідовність виразів розглядається як один вираз; його значенням є значення останнього виразу. Операція послідовного обчислення дозволяє на місці одного виразу записати кілька.

Операторам циклів с параметром **for** потрібно віддати перевагу при організації циклів з лічильниками.

В циклі **for** можна використовувати інструкції **break** та **continue.** Інструкція **break** у тілі циклу **for** завершує його виконання, а інструкція **continue** завершує виконання лише тіла циклу; відразу після неї виконується перехідна дія.

**Приклад.** Дуже часто інструкція циклу **for** зустрічається у вигляді

**for (k=0; k<n; ++k) *інструкція***

й задає виконання *інструкції* за значень *k* = 0, 1, 2, …, *n*-1 або у вигляді

**for (k=1; k<=n; ++k) *інструкція***

й задає виконання *інструкції* за значень *k* = 1, 2, …, *n*, або у вигляді

**for (k=n; k>0; --k) *інструкція***

й задає виконання *інструкції* за значень *k* = *n*, *n*-1, …, 2, 1. Змінну **k** у цих ситуаціях інколи називають **лічильником циклу**.

**Переривання break та продовження циклу continue**

Виконання інструкції **break** всередині циклу будь-якого різновиду перериває й завершує цикл; далі виконуються дії, наступні за цим циклом. Якщо **break** записано в інструкції циклу, вкладеній в іншу інструкцію циклу, то виконання **break** завершує вкладений цикл, а зовнішній цикл продовжується.

Інструкція **continue** всередині циклу задає перехід на кінець тіла циклу. В інструкціях циклу з перед- і післяумовою після **continue** обчислюється умова продовження циклу.

Приклад.За допомогою клавіатури вводиться послідовність дійсних чисел. Потрібно підрахувати суму її додатних елементів, а за появи 0 видати накопичену суму й завершити роботу.

Запрограмуємо цикл, в якому вводиться й обробляється послідовність чисел. Уведене число зберігаємо в змінній **x**, а суму додатних елементів – у змінній **sum**. Якщо під час уведення трапилася помилка, то подальші дії з уведення не виконуються, а змінна **x** зберігає своє останнє значення. Тому умовою продовження циклу буде саме відсутність помилок (інакше можна отримати цикл, який ніколи не завершиться!). Цю умову задає значення виразу введення **cin>>x**, перетворене до логічного типу.

**#include <iostream>**

**using namespace std;**

**int main()**

**{ double x;**

**double sum=0;**

**cout<<"Enter reals:\n";**

**while (cin>>x){**

**if (x==0.) break; //виходимо з циклу**

**if (x<0.) continue; //пропускаємо від'ємні**

**sum+=x;**

**}**

**cout << "sum=" << sum << endl;**

**system("pause");**

**return 0;**

**}**

Використання інструкції **continue** в цій програмі є дуже штучним. Ще одним недоліком є те, що в кінці не повідомляється, чи були помилки під час уведення. Інструкції програми виконуються *в порядку їх запису в програмі*. Про таку програму кажуть, що вона **структурована**. Інструкції **break** і **continue** *порушують* цей порядок обчислень, заплутуючи текст програми. Тому, користуючися ними, програміст повинен ретельно відслідковувати точку програми, якою продовжуються обчислення. Інколи ці інструкції дійсно скорочують запис розгалужень у циклі, проте в більшості випадків ті ж самі дії *можна описати без них*. Тому краще не зловживати **break** і **continue**.

**Збільшення та зменшення**

У циклічних обчисленнях дуже часто використовуються присвоювання вигляду **x=x+1** та **x=x-1**. Їх можна задати в скороченій формі за допомогою одномісних операторів **збільшення** (інкременту) **++** і **зменшення** (декременту)º**--**. Ці оператори (і відповідні операції) мають **префіксну (++x, --x**) і **постфіксну** (**x++, x--**) **форми**.

Вираз із постфіксним оператором **x++** або **x--** змінює значення змінної **x** на **1**, але значенням самого виразу є значення **x** *перед зміною*. Вираз із префіксним оператором **++x** або **--x** теж змінює **x** на **1**, але значенням виразу є значення **x**, отримане *після зміни*. Ці відмінності виявляються, коли оператори **++** та **--** застосовуються всередині виразів.

Операції **++** та **--** виконуються швидше ніж відповідні присвоювання вигляду **x=x+1** та **x=x-1**, тому рекомендується використовувати саме їх. Операції **++** та **--** застосовні до змінних будь-якого з базових типів, хоча найчастіше їх використовують із цілими змінними.

Скрізь, де немає необхідності використовувати старе значення змінної, рекомендується з виразів вигляду **n++** та **++n** вибирати **++n**, оскільки він виконується швидше й простіше.

Спосіб і порядок обчислення виразу залежить від компілятора, тому краще записувати операції збільшення або зменшення в окремих виразах або інструкціях, а не у складі інших виразів. Наприклад, значення виразів **(n++)\*(n++)** та **(++n)\*(++n)** у різних системах програмування навіть можуть відрізнятися. Гарантовано лише те, що до значення змінної **n** двічі додається 1.

**Принципи структурного програмування**

Структурне програмування це методологія й технологія розробки програмних комплексів, заснована на наступних принципах:

*- програмування* повинне здійснюватися зверху-*униз;*

- увесь проект повинен бути розбитий на *модулі* з одним *входом* і одним *виходом* (оптимальний розмір модуля — кількість *рядків* на екрані *дисплея);*

- логіка *алгоритму* й *програми* повинна допускати тільки три основні структури: *послідовне виконання, розгалуження* й *повторення.* Неприпустимий *оператор* передачі керування в будь-яке місце *програми;*

- при розробці *документація* повинна створюватися одночасно із *програмуванням,* у вигляді коментарів до програми.

Ціль структурного програмування — підвищення надійності програм, забезпечення *супроводу* й модифікації, полегшення й прискорення розробки. У програмах з використанням структурного програмування добре простежується основний алгоритм, вони більш зручні в налагодженні і менш чутливі до помилок програмування. Ідеї структурного програмування з'явилися на початку 70х рр. XX в. у компанії *IBM,* у їхній розробці брали участь відомі вчені Є. Дейкстра, X. Милі, Є. Батіг, С. Хоор.

Існує декілька важливих моментів структурного програмування:

1. Вихідний код повинен мати модульну структуру. Тобто, програма розділяється на дрібніші одиниці – процедури і функції. Ці частини або підпрограми, можуть викликатися з будь-якого місця у ній. Процедури – окремі ділянки коду, які виконують певні дії, задані алгоритмом та мають власну назву. Функції також можуть обчислювати деякі змінні,  мають значення, яке повертається, і можуть використовуватись в основній частині програми і в інших підпрограмах. Деякі підпрограми можуть мати рекурсивну структуру, тобто виклик з «самої себе». Це може допомогти вирішити задачу, але і призвести до зациклювання.
2. Кодування програми повинно виконуватися зверху-вниз чи знизу вгору. Схема «зверху-вниз» добре зрозуміла для дослідження написаної програми і пошуку помилок. Схема «знизу-вгору» використовується, коли алгоритм програми не розроблений, але вже написані деякі підпрограми, які реалізують певні дії.
3. Наявність керуючих елементів. У структурному підході використовуються цикли, умови і послідовності.

Структурне програмування поліпшило загальне сприйняття коду та сприяло легшому написанню програм. За методами структурного програмування, алгоритм програми стає універсальним і за рахунок цього будь-який інший розробник зможе його змінити або використати у своїй програмі.

**Проектування програми**.

Найчастіше алгоритм створюють, поступово уточнюючи поняття, пов'язані із задачею, і необхідні дії. Тоді кажуть, що розробку ведуть **згори донизу**.

Напишемо програму, що розв'язує рівняння *ax*+*b* =0.

**Уточнення постановки задачі**. Визначимо вхідні й вихідні дані програми. *Вхід*: коефіцієнти рівняння – два дійсних числа *a* та *b*. *Вихід*: кількість розв'язків; якщо розв'язок один – то саме цей розв'язок.

**Математичний аналіз задачі**. За умови *a* ≠0 рівняння має один розв'язок - *b*/*a*.

Тому потрібна перевірка, що вхідні дані коректні, тобто, що *a* ≠0.

**Проектування програми**.

У загальному вигляді алгоритм такий:

1. Отримати вхідні дані.

2. Обробити вхідні дані.

3. Вивести результат обробки.

Уточнимо кожен із кроків алгоритму.

"Отримати вхідні дані".

1.1. Вивести запрошення на введення даних.

1.2. Увести коефіцієнти рівняння в дійсні змінні **a** та **b**.

"Обробити вхідні дані". На основі аналізу задачі, якщо *a* ≠ 0, то кількість розв'язків дорівнює 1, а розв'язком є -*b*/*a*. Розв'язок запам'ятаємо в дійсній змінній **x**.

"Вивести результат обробки".

3.1. Вивести рівняння, уведене користувачем.

3.2. За допомогою значень змінної **x** вивести розв'язок, якщо він є, інакше вивести повідомлення про відсутність рішення.

Нарешті, можна кодувати.

**//програма, що розв'язує рівняння ax+b=0**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**double a=0, b=0; //** коефіцієнти рівняння

**double x; // розв'язок рівняння**

**// отримати вхідні дані**

**cout<<"Enter coefficients a and b of " << "equation ax+b=0 (2 reals)\n";**

**cin>>a>>b;**

**// Підготовити виведення результату**

**cout<<"Equation "<<a<<"x+"<< b <<"=0 has ";**

**// обробити введені дані**

**if (a!=0)**

**{x=(-b)/a;**

**// повідомити результат**

**cout <<" solution " << x<<"\n";**

**}**

**else cout<<" no solution\n"; // повідомити результат**

**system("pause");**

**return 0;**

**}**

У цій програмі кожен фрагмент коду задає певні дії для отримання необхідного результату, тобто має своє призначення, або свій **обов'язок**. На перший погляд, програму можна зробити коротшою: якби обчислювати й відразу виводити розв'язок. Однак тоді код обробки даних був би *перевантажений обов'язками*, тобто відповідав за кілька різних функцій (тут – обчислення й виведення на екран).

Якщо кожен фрагмент коду має своє, персональне призначення, то це, по-перше, робить загальну структуру програми прозорішою і, по-друге, полегшує модифікацію окремих частин програми. У наведеному прикладі можна забажати змінити вихідне текстове повідомлення, і це не вплине на алгоритм обчислення результату. Отже, відокремлення обробки від виведення результатів цілком обґрунтоване.

Ускладнимо розглянуту задачу. Нехай розв’язок потрібно знайти не для однієї пари чисел *a* та *b*, а для якоїсь заздалегідь невідомої кількості їх комбінацій. В цьому випадку значення *a* та *b* потрібно вводити в циклі та запитувати можливість припинення. Для цього програма може бути модифікована таким чином.

**//програма, що розв'язує рівняння ax+b=0 для різних пар a,b**

**#include <iostream>**

**using namespace std;**

**int main()**

**{**

**double a=0, b=0; // коефіцієнти рівняння**

**double x; // розв'язок рівняння**

**int IsContinue=1; // Ознака продовження**

**while (IsContinue == 1)**

**{**

**// отримати вхідні дані**

**cout<<"Enter coefficients a and b of " << "equation ax+b=0 (2 reals)\n";**

**cin>>a>>b;**

**// Підготовити виведення результату**

**cout<<"Equation "<<a<<"x+"<< b <<"=0 has ";**

**// обробити введені дані**

**if (a!=0)**

**{x=(-b)/a;**

**// повідомити результат**

**cout <<" solution " << x<<"\n";**

**}**

**else cout<<" no solution\n"; // повідомити результат**

**cout <<" Enter 1 for continue \n ";**

**cin>> IsContinue;**

**}**

**system("pause");**

**return 0;**

**}**

*Для самостійного вивчення (2 години)*: Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.

*Література*

1. Ковалюк Т. В. Алгоритмізація та програмування: Підручник. — Львів: «Магнолія 2006», 2013. — 400 с., ил.
2. Єжова Л. Ф. Алгоритмізація і програмування процедур обробки інформації: Навч.-метод. посіб. для самост. вивч. дисц. К.: КНЕУ, 2000.
3. Вступ до програмування мовою С++. Організація обчислень: навч. посіб. / Ю. А. Бєлов, Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставовський. – К. : Видавничо-поліграфічний центр "Київський університет", 2012. – 175 с. с.: іл. ISBN (укр.) . URL: <http://csc.knu.ua/uk/library/books/belov-24.pdf>
4. Джейс Либерти Освой самостоятельно С++ за 21 день: 3-е изд. пер. с англ.: Уч. пос. – М.: Издательский дом „Вильямс”, 2001. – 816 с.: ил..

*Контрольні запитання для самоперевірки*.

1. Які оператори циклу використовуються у C++?
2. Як працює оператор циклу for?
3. Пояснити на прикладах використання циклу з передумовою і циклу з післяумовою.
4. Які оператори з розглянутих є операторами керування у мові C++?
5. Наведіть власний приклад оператор-перемикача **switch**.
6. Наведіть власні приклади всіх форм оператору циклу.
7. Наведіть власні приклади використання операторів переривання **break** та продовження циклу **continue.**